Automatically Adjusting Concurrency to the Level of Synchrony

Pierre Fraigniaud, Eli Gafni, Sergio Rajsbaum, Matthieu Roy

Automatically Adjusting Concurrency to the Level of Synchrony*

Pierre Fraigniaud[1**], Eli Gafni[2], Sergio Rajsbaum[3 ***], and Matthieu Roy[4]

[1] CNRS and University Paris Diderot, France
[2] Dpt. of Computer Science, UCLA, USA
[3] Instituto de Matemáticas, UNAM, Mexico.
[4] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France

Abstract. The state machine approach is a well-known technique for building distributed services requiring high performance and high availability, by replicating servers, and by coordinating client interactions with server replicas using consensus. Indulgent consensus algorithms exist for realistic eventually partially synchronous models, that never violate safety and guarantee liveness once the system becomes synchronous. Unavoidably, these algorithms may never terminate, even when no processor crashes, if the system never becomes synchronous.
This paper proposes a mechanism similar to state machine replication, called *RC-simulation*, that can always make progress, even if the system is never synchronous. Using RC-simulation, the quality of the service will adjust to the current level of asynchrony of the network — degrading when the system is very asynchronous, and improving when the system becomes more synchronous. RC-simulation generalizes the state machine approach in the following sense: when the system is asynchronous, the system behaves as if $k+1$ threads were running concurrently, where k is a function of the asynchrony.
In order to illustrate how the RC-simulation can be used, we describe a long-lived renaming implementation. By reducing the concurrency down to the asynchrony of the system, RC-simulation enables to obtain renaming quality that adapts linearly to the asynchrony.

1 Introduction

Problem statement. The *state machine* approach (also called *active replication*) [33,37] is a well-known technique for building a reliable distributed system requiring high performance and high availability. In the state machine approach, a *consensus* algorithm is used by the replicas to simulate a *single* centralized state machine. The role of consensus is to ensure that replicas apply operations to the state machine in the same order. Paxos [34] is the most widely-used

* This work is supported in part by the CONACYT-CNRS ECOS Nord M12M01 research grant, by CNRS-PICS and the RTRA STAE. Part of this work was done while the first three authors were visiting LAAS in Toulouse, France, July 2013.
** Additional supports from ANR project DISPLEXITY and INRIA project GANG.
*** Additional support from UNAM-PAPIIT

consensus protocol in this context. It maintains replica consistency even during highly asynchronous periods of the system, while rapidly making progress as soon as the system becomes stable.

The state machine approach is limited by the impossibility of solving consensus in an asynchronous system even if only one process can crash [22]. Indulgent [26] consensus algorithms such as Paxos, never violate safety (replicas never decide different values), and they guarantee liveness (all correct replicas decide) once the system becomes synchronous [32]. However, while the system is in an asynchronous period, the state's machine progress is delayed. Moreover, any indulgent consensus algorithm has executions that never terminate, even when no processor crashes, due to the impossibility of [22], in case the system never becomes synchronous.

One may think that to achieve reliability in distributed systems, and to enforce cooperation among the replicas enabling the system to function as a whole despite the failure of some of its components, consensus is essential [23]. This is true in general, but not always. E.g., consensus is not essential for implementing replicated storage [7] (the dynamic case is discussed in [3]). Hence, the question of whether one can build a reliable distributed system that always makes progress, for some specific services at least, remains open. This is precisely the question we are interested in.

Summary of results. In this paper, we provide a mechanism similar to state machine replication, but that can always make progress, even if the system is never synchronous. Using our mechanism, the quality of the service adjusts to the current level of asynchrony of the system – degrading when the system is very asynchronous, and improving when the system becomes more synchronous. The main contribution of this paper is the proof that such a mechanism exists. We call it the *reduced-concurrency simulation* (RC-simulation for short).

To be able to design such a mechanism, we had to come up with appropriate definitions of "quality" of a service, and of the "level of asynchrony" of the system. In the state machine approach, the service behaves as one single thread once the system becomes synchronous. This behavior is generalized through the RC-simulation, so that, when the level of asynchrony is k, then the system behaves as if $k + 1$ threads were running concurrently.

In order to illustrate, with a concrete example, how the RC-simulation is used to obtain a fault-tolerant service that always makes progress, we describe a long-lived renaming service. In this case, the quality of the service is manifested in terms of the output name space provided by the long-lived renaming service, the smaller the better. Thanks to the RC-simulation mechanism, a higher quality of service is obtained by reducing the concurrency down to the level of asynchrony of the system. In particular, if the system is synchronous, then the service behaves as if names were produced by a single server, giving to the clients names from a very small (optimal) space. If the system becomes more asynchronous, then the service behaves as if more servers were running concurrently, and hence it gives names from a larger space to the clients. Whatever the asynchrony of

the system is, safety is never violated, in the sense that the names concurrently given to the clients are always pairwise distinct.

The formal setting we consider for deriving our results is the one of an asynchronous read/write shared memory system where any number of processes may fail by crashing. We are interested in *wait-free* algorithms [27]. For simplicity, we assume that snapshot operations are available, since they can be implemented wait-free [1], and we define level of asynchrony at this granularity. However, the definition can be easily adapted to the granularity of read/write operations. We also stress the fact that we picked this shared memory model, and we picked the specific renaming textbook algorithm in [9], as a proof of concept. Definitely, further work is needed to study other, more realistic shared memory and/or message passing models, as well as other services. Moreover, we have not tried to optimize the *efficiency* of our simulation, which is beyond the objective of this paper. The main outcome of the paper is a proof that it is *possible* to adapt the concurrency to the level of asynchrony in a generic way.

Related work. Various partially synchronous models have been proposed to model real systems better. For instance, in [19,21], the relative processors speeds are bounded, and there are bounds on processing times and communication delays. In addition, some of these models allow the system to have an initial period where the bounds are not respected. However, it is assumed that, eventually, the system enters a *stable* period where the bounds do hold. More recently, partially synchronous systems enabling to study problems that are strictly weaker than consensus were considered in [2]. Also, there is work on progress conditions that adapt to the degree of synchrony in each run, e.g. [4] and references herein.

Although RC-simulation tackles different objectives, it is inspired by the *BG-simulation algorithm* [12]. The latter is used for deriving reductions between tasks defined in different models of distributed computing. Indeed, the BG-simulation allows a set of $t + 1$ processes, with at most t crash failures, to "simulate" a larger number n of processes, also with at most t failures. The first application of the BG-simulation algorithm was that there are no k-fault-tolerant algorithms for the n-process k-set-agreement problem [17], for any n. Borowsky and Gafni extended the BG-simulation algorithm to systems including set agreement variables [11]. Chaudhuri and Reiners later formalized this extension in [18], following the techniques of [36]. While the original BG-simulation works only for colorless tasks [12] (renaming and many other tasks are not colorless [14,29]), it can be extended to any task [24]. Other variants and extensions of the BG-simulation, together with additional applications, have appeared in [16,28,30,31,35].

At the core of the BG-simulation, and of our RC-simulation as well, is the *safe-agreement* abstraction, a weaker form of consensus that can be solved wait-free (as opposed to consensus). The safe-agreement abstraction was introduced in [10] as "non-blocking busy wait." Several variants are possible, including the wait-free version we use here (processes are allowed to output a special value $\perp$ that has not been proposed), and the variant that was used in the journal version [12]. Safe-agreement is reminiscent of the crusader problem [20]. A

safe-agreement extension that supports adaptive and long-lived properties is discussed in [6]. Notice that we can apply our RC-simulation to renaming, which is not a colorless task (and hence cannot be used with the BG-simulation), because we use safe-agreement only to simulate threads consistently, as opposed to the BG-simulation, which uses safe-agreement for reductions between tasks.

A generalization of the state machine approach was presented in [25], where, instead of using consensus, it uses set agreement. In k-set agreement, instead of agreeing on a unique decision, the processes may agree on up to k different decisions. The paper [25] presents a state machine version with k-set agreement, where any number of processes can emulate k state machines of which at least one remains "highly available". This state machine version is not based on the BG-simulation, and is not dynamic, as opposed to our RC-simulation mechanism.

Renaming is a widely-studied problem that has many applications in distributed computing (see, e.g., the survey [15]). In long-lived renaming, the number of processes participating in getting or releasing a name can vary during the execution. When the number is small, the number of names in use should be small. An $f(k)$-renaming algorithm [8] is a renaming algorithm in which a process always gets a name in the range $\{1, \ldots, f(k)\}$, where k is the number of processes that participate in getting names. Notice that $f(k)$-renaming is impossible unless $f(k) \geq k$. Burns and Peterson [13] proved that long-lived $f(k)$-renaming is impossible in an asynchronous shared memory system using only reads and writes unless $f(k) \geq 2k - 1$. They also gave the first long-lived $(2k - 1)$-renaming algorithm in this model. for our example is from [9]. Finally, recall that indulgent algorithms progress only when synchrony is achieved. We refer to [5] for sensing fast that synchrony is on, and for solving renaming with an indulgent algorithm.

2 Reducing Concurrency

In this section, we describe the *reduced-concurrency* simulation, abbreviated in *RC-simulation* hereafter. At the core of the RC-simulation is the *safe-agreement mediated* simulation, or *SA-simulation* for short. The SA-simulation is the essential mechanism on top of which is implicitly built the well known BG-simulation [12]. We describe the SA-simulation explicitly in Section 2.2, and then explain in Section 2.3 how the RC-simulation is built on top of the SA-simulation in order to reduce concurrency. Hence, the contribution of this section is the complete description of the plain arrow on the left-hand side of Fig. 1, involving the notion of *partial asynchrony*, that we define next, in Section 2.1.

2.1 Partial Asynchrony

We are considering a distributed system composed of n crash-prone asynchronous processes. Each process p has a unique identity $Id(p) \in \{1, \ldots, n\}$. Processes communicate via a reliable shared memory composed of n multiple-reader single-writer registers, each process having the exclusive write-access to one of these

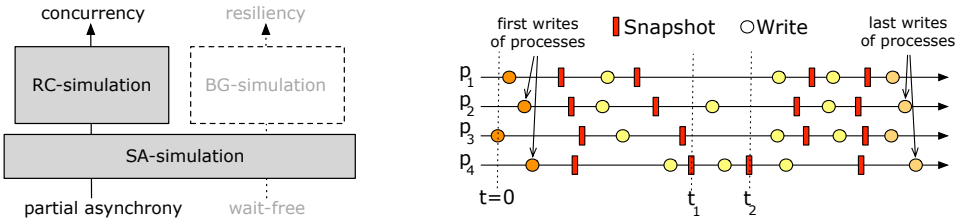

Fig. 1. RC- vs. BG-simulations (left), illustration of partial asynchrony (right)

n registers. For simplicity, by "read", we will actually mean an *atomic snapshot* of the entire content of the shared memory (recall that snapshots can be implemented wait-free using read/write registers [1]).

In the wait-free setting, the efficiency of the system can be altered by the asynchrony between the processes. To measure the amount of asynchrony experienced by the system, we introduce the novel notion of *partial asynchrony*. (See Fig. 1 for an illustration of the concepts defined hereafter). Let us consider a potentially infinite execution $\mathcal{E}$ of the system, and let p be a process.

A *blind interval* for p is a maximal time-interval during which p performs no snapshots. More precisely, let t_1 and t_2 with $t_2 > t_1$ be the times at which two consecutive snapshots performed by p occur in $\mathcal{E}$. The time-interval $I = (t_1, t_2)$ is called a blind time-interval for process p. If in addition we assume, w.l.o.g., that the first instruction of every process is a write, then the interval $[0, t)$ where t denotes the time at which process p performs its first snapshot, is also a blind interval for p. Thus, $\mathcal{E}$ can be decomposed into the sequence of consecutive blind intervals for a process p, where a snapshot by p is performed in between each of these intervals. In Fig. 1, each process performs four snapshots and five writes. Hence, there are five blind intervals for every process.

Definition 1. *The* partial asynchrony *experienced by process p in $\mathcal{E}$ during a blind interval for p is the number of processes that do not perform any write during this time-interval. The partial asynchrony of p in $\mathcal{E}$ is the maximum, taken over all blind intervals I for p, of the partial asynchrony experienced by process p during I. Finally, the partial asynchrony of the system in $\mathcal{E}$ is the maximum, taken over all processes p, of the partial asynchrony of p in $\mathcal{E}$.*

On the right-hand side of Fig. 1, p_1 experiences a partial asynchrony of 1 because, p_4 is missing in the interval between its first two snapshots. The partial asynchrony of the whole execution is 2 because p_4 experiences a partial asynchrony of 2 in its blind interval (t_1, t_2) as both p_1 and p_3 are missing, but no processes experience a partial asynchrony of 3. For any non-negative integer k, an execution in which the partial asynchrony of the system is at most k is called k-partially asynchronous. In particular, an execution is 0-partially asynchronous if and only if all processes perform in lock steps. As another example, if the process p is much slower than all the other processes, then p experiences zero

partial asynchrony. However, each of the $n-1$ other processes experiences a partial asynchrony of 1 caused by the slow process p. The general case is when the processes are subject to arbitrary delays, as illustrated in the example of Fig. 1. Then the larger the partial asynchrony experienced by a process p, the more asynchronous the system looks to p, and the more p may suffer from this asynchrony. Our objective is to limit the penalty incurred by the system because of asynchrony, by reducing the amount of concurrency.

2.2 The Safe-Agreement Mediated Simulation

The *safe-agreement mediated* simulation, or *SA-simulation* for short, is a generic form of execution of a collection $\mathcal{T}$ of *threads* by a system of n processes. Hence, let us first define the notion of thread.

Threads. A thread τ is a triple $(Id, A, value)$ where $Id = Id(\tau)$ is the *identity* of the thread τ, uniquely defining that thread, $A = A(\tau)$ is a set of instructions to be performed by the thread (typically, this set is the one corresponding to the code of some process executing a distributed wait-free algorithm), and $value = value(\tau)$ is an input value to this instruction set. The generic form of a thread code is described in the left-hand side of Fig. 2, where φ is a boolean predicate, and f is a function, both depending on A. In this generic form, a thread is thus a write, followed by a finite sequence of snapshot-write instructions. In the sequel, when we refer to an *instruction* of a thread, we precisely refer to such a snapshot-write instruction. The SA-simulation is an execution by n processes of the threads in a collection $\mathcal{T}$ of threads.

```
THREAD τ = (Id, A, value)
1:  view ← (Id, value)
2:  write(view)
3:  while φ(view) do
4:      view ← snapshot()
5:      write(view)
6:  decide f(view)
```

```
SA-SIMULATION of a collection T of threads.
1:  while true do
2:      view ← snapshot()
3:      if there exists an extendable thread τ ∈ T
4:      then perform next instruction of τ
```

Fig. 2. Generic code for threads (left), and code for SA-simulation (right)

For the sake of simplicity of the presentation, we assume that the threads are resident in memory when the simulation begins. Moreover, we also assume that each thread is either *available* or *non available*. Our simulation is concerned with simulating available threads. However a thread can be moved from non-available to available (not vice-versa), but such a move is not under the control of the system, and can be viewed as under the control of an adversary. This models the situation in which the system receives requests for executing threads, or for performing tasks, by abstracting away the input/output interface of the system

with the outside world. Taking once again the example of long-lived renaming, a thread corresponding to the release of a name by a process can be made available only if a name has been acquired previously by this process, and two requests of names by a same process must be separated by a release name by the same process. Other than that, the adversary is entirely free to decide when processes request and release names.

SA-simulation in a nutshell. In essence, the SA-simulation relies on two notions: *extendability* and *agreement*. Roughly, a thread is *extendable* if it is not blocked, that is, if no processes are preventing others from performing the next instruction of the thread. Indeed, again roughly, performing an instruction of a thread requires some form of coordination among the processes, in order to decide which value to write, that is, to decide which view of the memory is consistent with a view that may have acquired a thread by snapshotting the memory. Since consensus is not possible to achieve in a wait-free environment [22], we will use weaker forms of agreement.

Let us present the general form of SA-simulation. The code of a process p performing SA-simulation of $\mathcal{T}$ is presented in Fig. 2. We assume a long-lived simulation, i.e., an infinite number of threads. Handling a finite number of threads just requires to change the condition in the while-loop of the SA-simulation, in order to complete the simulation when all (available) threads have been simulated. In what follows, we precisely explain how a process figures out whether there exists an extendable thread, and how it performs the next instruction of a thread. For this purpose, we first recall a core mechanism for performing a weak form of consensus: *safe-agreement*.

SAFE-AGREEMENT performed by process p proposing $v = \mathit{value}(p)$

1: write $(\mathit{Id}(p), v)$
2: snapshot memory, to get $\mathit{view} = \{(i_1, v_{i_1}), \ldots, (i_k, v_{i_k})\}$
3: write $(\mathit{Id}(p), \mathit{view})$
4: snapshot memory, to get $\mathit{setofview} = \{(j_1, \mathit{view}_{j_1}), \ldots, (j_\ell, \mathit{view}_{j_\ell})\}$
5: $\mathit{minview} \leftarrow \bigcap_{r=1}^{\ell} \mathit{view}_{j_r}$
6: **if** for every i such that $(i, v_i) \in \mathit{minview}$ we have $\mathit{view}_i \in \mathit{setofview}$
7: **then** decide minimum value w in $\mathit{minview}$
8: **else** decide $\perp$

Fig. 3. Safe agreement

Safe-agreement. Consider the following specification of *safe-agreement* (similar to [12]). Each process proposes a value, and must decide some output according to the following three rules: (1) *Termination*: every process that does not crash must decide a proposed value or $\perp$; (2) *Validity*: if not all processes crash,

then at least one process must decide a value different from $\perp$; (3) *Agreement*: all processes that decide a value different from $\perp$ must decide the same value. The algorithm in Fig. 3 is a wait-free algorithm solving safe agreement, directly inspired from [6].

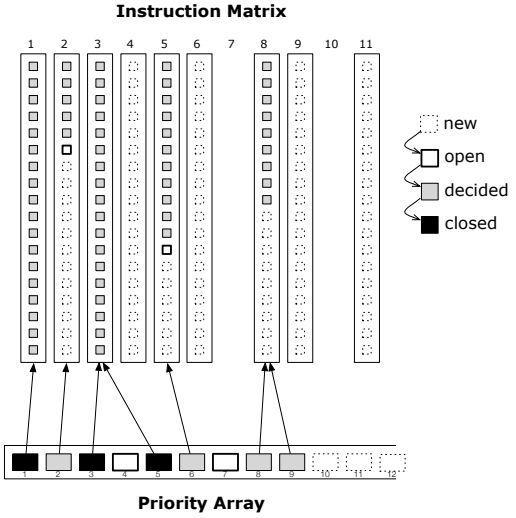

Fig. 4. *Virtual data structure maintained at each node*

The instruction matrix. We explain now how processes decide whether there exist extendable threads. This is achieved thanks to the *virtual* shared data-structure displayed beside. Whenever a process p performs a snapshot of the memory, it gets a view of the current states of execution of all the threads. This view can be virtually represented by the instruction matrix M as displayed in Fig. 4. The cell $M_{i,j}$ of the instruction matrix represents the status of the ith instruction of the jth thread (i.e., the thread τ such that $Id(\tau) = j$). This status can be one of the following three states: (1) new: no processes have tried to perform this instruction yet, (2) open: at least one process is trying to perform this instruction, but the outcome of this instruction has not yet been decided, or (3) decided: the instruction has been performed, that is, an outcome has been decided for this instruction. A thread for which all cells are decided is said to be *fully executed* in the sense that all its instructions have been performed by the system. Of course, the SA-simulation executes the instructions of every thread in sequential order. That is, the column j of the instruction matrix corresponding to a thread that is available but not fully executed, consists in a (possibly empty) sequence of decided cells, immediately followed by a cell that is either open or new, immediately followed by a (possibly empty) sequence of cells in state new. The cell immediately following the (possibly empty) sequence of decided cells is called the *current instruction cell* of the thread. The instruction matrix enables to define formally the notion of extendability:

Definition 2. *A thread τ is* extendable *if it is available, not fully executed, and its current instruction cell is in state new.*

Note that the instruction matrix is an abstraction describing the execution status of the threads. It is not a data structure that is manipulated by the processes. Indeed, when a process performs the snapshot in SA-simulation, its view is the content of the whole memory, which contains the way the threads are progressing, including the number of snapshot-write instructions that have been already performed by each thread. In the SA-simulation, every process identifies some extendable thread τ, and aims at performing the next instruction of τ. Roughly, when a process tests whether there exists an extendable thread in SA-simulation, it tests all open cells in the instruction matrix using safe-agreement, and the answer is negative if and only if this process has decided $\perp$ for each of them. This mechanism is detailed next.

Performing instruction. Recall that what we defined as an "instruction" of a thread is a snapshot followed by a write. Such a combination of snapshot-write appears explicitly in the generic form of a thread (cf. Fig. 2). The value written in memory by a thread is simply the view it gets during the last snapshot preceding the write. Hence, to simulate an instruction of a thread, the potentially many processes performing that instruction must agree on this view. This coordination is performed via safe-agreement.

There is one safe-agreement task associated to each cell (i, j) of the instruction matrix, $i, j \geq 1$. A process p aiming at simulating Instruction i of Thread j proposes its view to the safe-agreement of cell (i, j). By the specification of safe-agreement, it is guaranteed that, among all views proposed to this safe-agreement, only one of them with be outputted. (Not all processes may decide this view, but those ones decide $\perp$). If p outputs $\perp$, then p aborts this attempt to perform the instruction, and completes the while-loop of SA-simulation. Otherwise, i.e., if p outputs some view, then p writes this view for Thread j, to complete the write of that thread.

Safety and Liveness. We now establish the following lemma, which may have its interest on its own, but will also be later used to prove the correctness of our more refined RC-simulation. Let us define two crucial properties that are expected to be satisfied by a simulation of threads by a system of processes.

Safety: For every collection $\mathcal{T}$ of threads solving task T using a same algorithm A, the simulation of $\mathcal{T}$ by SA-simulation leads $\mathcal{T}$ to solve Task T.

Liveness: In any infinite execution of the SA-simulation on a finite collection $\mathcal{T}$ of threads in which t processes crash, with $0 \leq t < n$, all but at most t threads are fully executed.

Note that, in the definition of safety, the specification of a thread may depend on other threads: the consistency of the simulation is global. Also, the definition of liveness states that if t processes crash then at most t threads may

be "blocked" by these t failures. Note however that the SA-simulation does not necessarily guarantee that all but t threads will eventually be executed if the collection of threads is infinite. Indeed, the SA-simulation does not avoid starvation. Starvation is a property addressed at a higher level of the simulation, depending on the policy for choosing which threads to execute among all extendable threads on top of the SA-simulation. For instance, BG-simulation selects the threads in an arbitrary manner, but executes these threads in a specific round-robin manner. Instead, RC-simulation, described later in this section, executes the threads in an arbitrary manner, but selects these threads in a specific manner to minimize concurrency. The following result states the basic properties of the SA-simulation.

Lemma 1. *The SA-simulation satisfies both the safety and liveness properties.*

2.3 The Reduced-Concurrency Simulation

In the SA-simulation, all available threads are potentially executed concurrently. We now show how to refine the simulation in order to execute concurrently as few threads as possible, while still remaining wait-free. For this purpose, we first refine the notion of available threads, by distinguishing two kinds of available threads. An available thread is *active* if it is not fully executed, and at least one cell of the column corresponding to that thread in the instruction matrix is either open or decided. An available thread that is not fully executed and not active is *pending*. In the SA-simulation, every process tries to make progress in any available extendable thread, including pending threads. Instead, in the RC-simulation described below, every process tries to make progress only for active extendable threads. It is only in the case where there are no more active extendable threads that the RC-simulation turns a pending thread into active. In this way, the number of active threads is not blowing up. Reducing the concurrency as much as possible however requires more work, and will be explained in detail in this section. One key ingredient in the RC-simulation is the *Priority Array*, described below.

The Priority Array. Similarly to the Instruction Matrix, the Priority Array is a *virtual* shared data-structure that enables to summarize the status of the different threads, as well as an instrument used to decide which thread to execute, and when (see the figure in Section 2.2). It is a linear array, in which the role of each cell is to point to a thread. As in the instruction matrix, there is a safe-agreement task associated to each cell of the priority array. This safe-agreement can be in one of the four following states: new, open, decided, and closed. For $i \geq 1$, Cell i of the priority array is *new* if no processes have yet entered the safe-agreement program i (i.e., no processes have yet performed the first instruction of the ith Safe-Agreement). A cell is *open* if at least one process has entered Safe-Agreement i, but no value has yet been decided (all processes that exited the safe-agreement decided $\perp$). A cell is *decided* if at least one process has decided

a value (different from $\perp$). Such a value is the identity of an available thread. Hence, a decided cell in the priority array is a pointer to an available thread. By construction, an active thread is an available thread pointed by a decided cell of the priority array. Finally, Cell i is *closed* if it is pointing to a fully executed thread. Initially, all cells of the priority array are in state new. The *head* of the priority array is defined as the cell with smallest index that is still in state new. We now have all the ingredients to describe the RC-simulation.

The RC-simulation. The code of the RC-simulation is described in Fig. 5. The first instructions of RC-simulation are essentially the same as in the SA-simulation, with the unique modification that thread-instructions are only performed in active threads, while the SA-simulation is willing to advance any extendable thread.

```
RC-SIMULATION by processes p of an infinite collection 𝒯 of threads.
1:  while true do
2:      view ← snapshot()
3:      (ℳ, 𝒫) ← extract instruction matrix and priority array from view
4:      if there exists an active extendable thread τ ∈ 𝒯
5:      then perform next instruction of τ
6:      else view ← snapshot()
7:          (ℳ′, 𝒫′) ← extract instruction matrix and priority array from view
8:          if (ℳ′, 𝒫′) = (ℳ, 𝒫), and there exists a pending thread
9:          then propose a pending thread to the head of the priority-array
```

Fig. 5. RC-simulation

Significant differences between the SA-simulation and the RC-simulation begin from the "else" Instruction. In particular, Instruction 8 compares the content of the memory at two different points in time. In the RC-simulation, a process p is willing to *open* a new thread, i.e., to move one available thread from the pending status to the active status, only if the memory has not changed between two consecutive snapshots of p (i.e., during a blind interval for p). More precisely, process p may open a new thread only if, in between two consecutive snapshots, the set of active threads has remained the same, and the instruction counters of these threads have remained the same. If that is the case, then p selects a thread τ among the pending threads (e.g., the one with smallest identity), and proposes τ for being opened. The role of the test in Instruction 8 will appear clearer later when we will analyze the number of threads that are simulated concurrently.

For opening a new thread τ, process p is proposing $Id(\tau)$ to the safe-agreement task at the head of the priority array. If p decides $\perp$ in this safe-agreement, then p initiates a new loop of RC-simulation. If p decides a value $j \neq \perp$ in this safe-agreement, then p tries to perform the first instruction of Thread j, which becomes active. Note that we may not have $j = Id(\tau)$ as other processes may have proposed different threads to this safe-agreement of the priority array. This

holds even if the rule for selecting which thread to propose is deterministic (e.g., selecting the pending thread with smallest identity) because of the asynchrony between the processes, and because of the way threads become available. As for the general case, performing the first instruction of Thread j is mediated by Safe-Agreement $(1, j)$ of the Instruction Matrix. After this attempt to perform the first instruction of Thread j, process p starts a new loop of RC-simulation, independently from whether this attempt succeeded or not.

2.4 The Reduced-Concurrency Theorem

We first note that the RC-simulation satisfies the same safety and liveness conditions as the SA-simulation.

Lemma 2. *The RC-simulation satisfies both the safety and liveness properties.*

Assume now that all the threads in the collection $\mathcal{T}$ simulated by RC-simulation are performing the same wait-free algorithm A solving some task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, such as renaming. Hence, the threads differ only in their identities, and, possibly, in their input data. The *performances* of Algorithm A, e.g., the number of names for renaming, may be affected by *point-contention*, i.e., by the maximum number of threads that are executed concurrently. For instance, a point-contention of 1 (that is, threads go one after the other) ensures perfect-renaming, while a point-contention of $k = |\mathcal{T}|$ may yield up to $2k - 1$ names. In this framework, RC-simulation helps, for it reduces the concurrency of the threads, down to partial asynchrony.

More specifically, let us consider an n-process system performing RC-simulation of a collection $\mathcal{T}$ of threads, all with same algorithm A solving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ wait-free. For every k, $0 \leq k < n$, if the partial asynchrony of the execution of the RC-simulation is at most k, then the performances of solving Task T mediated by the RC-simulation are the same as if the threads were executed with point-contention at most $k + 1$. This result can be summarized as follows.

$$\text{RC-simulation:} \quad k\text{-partial-asynchrony} \Longrightarrow (k+1)\text{-concurrency.}$$

In particular, if the partial asynchrony is 0 then threads are executed sequentially, one after the other. The following theorem formalizes these statements.

Theorem 1. *Let k, $0 \leq k < n$. For any execution of the RC-simulation:*

- Bounded concurrency: *Concurrency is at most partial-asynchrony plus 1. Specifically, if $k+1$ threads are executed concurrently by the system at time t, then at least one process p experienced a partial asynchrony at least k at some time $\leq t$.*
- Adaptivity: *Concurrency ultimately scales down with partial asynchrony. Specifically, if the partial asynchrony of the system is at most k after time t, then, eventually, the system will not execute more than $k + 1$ threads concurrently.*

Proof. To establish the theorem, we first observe a crucial property of the algorithm solving safe-agreement. This algorithm includes two write-instructions, and two snapshot-instructions. These instructions interleave in arbitrary manner among the different processes, but one can still say something interesting about this interleaving, in certain cases. The following behaviour is a common argument is the BG-simulation (see, e.g., [6,12]).

Claim 1. During the execution of the safe-agreement algorithm, if some process p decides $\perp$, then there exists a process $q \neq p$ such that q performed its first write before the first snapshot of p, and p performed its second snapshot before the second write of q.

In the situation of the statement of Claim 1, we say that q *blocked* p. Let us first prove the bounded concurrency property. Assume that $k+1$ threads are executed concurrently by the system at time t. At this time t, let $i \geq 1$ be the largest index such that the ith cell of the priority array is not new (that is, the cell just before the head of the priority array at time t). Let p be the first process that performed a write instruction in the safe-agreement corresponding to that cell, turning the status of that cell from new to open. Process p did so because the instruction matrix as well as the priority array remained in identical states between its two previous snapshots in the RC-simulation. Let $t' \leq t$ be the time at which process p wrote in the ith cell of the priority array, turning it from new to open. Let t_1 and t_2, $t_1 < t_2 < t' \leq t$, be the respective times at which p performed its two snapshots in Instructions 2 and 6 of the RC-simulation leading p to access Cell i.

Let us examine the instruction matrix and the priority array between time t_1 and t_2 (both did not change in this time interval). Let k' be the number of threads concurrently executed at time t_2. Between time t_2 and t, some new threads, say x threads, $x \geq 0$, may have appeared (i.e., moved from pending to active), while some of the k' threads, say y, $y \geq 0$, may have disappeared (i.e., moved from active to fully executed). The former x corresponds to cells in the priority array that move from open to decided. The latter y corresponds to cells in the priority array that move from decided to closed. By definition, we have $k' + x - y = k + 1$, and thus $k' + x \geq k + 1$. Among the x threads, one thread may have been opened by p, when p initiates the safe-agreement of Cell i in the priority array. Now, since there were no extendable threads at time t_1, each of the k' threads has the cell of the instruction matrix corresponding to its current instruction in state open. Therefore, during time interval $[t_1, t_2]$, the total number of open cells equals at least $k' + x - 1$, where k' are open in the instruction matrix, and at least $x - 1$ are open in the priority array. Thus, during time interval $[t_1, t_2]$, the total number of open cells is at least k since $k' + x \geq k + 1$. Each of these cells corresponds to a safe-agreement for which no values are decided. In particular, p decided $\perp$ from each of these safe-agreements.

Let us fix one of these at least k safe-agreements. By Claim 1, process p was blocked by some process q which has not yet performed its second write at time t_2. Thus, to each of the at least k open cells in interval $[t_1, t_2]$ corresponds

a distinct process which performed its first write (in the safe agreement of the cell) before time t_1, and had not yet performed its second write at time t_2. Hence, at least k processes performed no writes in the time interval $[t_1, t_2]$, which is the interval between two consecutive snapshots of process p. Therefore, the asynchrony experienced by process p at time $t_2 \leq t$ is at least k. This completed the proof of the bounded concurrency property.

Now remains to prove the adaptivity property. Let N be the number of threads that are concurrently executed at time t. By the same arguments as those exposed above, no threads are opened after time t until their number goes below $k+1$. By the liveness property of Lemma 2, the number of thread concurrently executed after time t will thus decrease from N down to $k+1$, which completes the proof of Theorem 1. □

3 Application to long-lived renaming

In this section, we use the RC-simulation for improving the performances of a classical long-lived renaming algorithm. Recall that the performances of an algorithm achieving renaming are typically measured in terms of the range of name-space, the tighter the better. In the context of this paper, the specification of long-lived renaming are rephrased as follows. Users perpetually request *names* to an n-process system such as described in Section 2.1, where a name is a non-negative integer value in the range $[1, N]$ for some $N \geq n$ that is aimed to be as small as possible. Each user's request is performed by any one of the n processes, and all processes are eligible to receiving requests from users. Once a user has acquired a name, the user must eventually release it, and, as soon as a name has been released, it can be given to another user. At any point in time, all names that are currently acquired by users, and not yet released, must be pairwise distinct. We assume that a process p serving the request for a name by a user x does not serve any other request for names until the requested name has been acquired by x, and eventually released. The textbook renaming algorithm described in [9] is an adaptation to shared-memory system of the classical algorithm of [8] for renaming in message passing systems. It implements renaming with new names in the range $[1, 2n-1]$. We show that, using RC-simulation, this range of names can be significantly reduced when the partial asynchrony is small, while the algorithm is shown not to adapt to partial asynchrony by itself.

Long-lived renaming in presence of bounded partial-asynchrony. In long-lived renaming, each process p requests a new name by invoking *ReqName*(x), where x denotes the identifier of the user which queried p to get a name. Once p got a name for x, it can call *releaseName* for releasing that name. According to the specification of long-lived renaming stated above, for any process p, a sequence of such calls is correct if and only if *ReqName* and *releaseName* alternate in the sequence. As mentioned before, the renaming algorithm we use provides names in the range $[1, 2n-1]$. We first note that this range does not reduce much as a function of the asynchrony. For example, Observation 1 below shows that, even in case of partial asynchrony zero, i.e., even when the system performs

in lock-steps, the range of names used by this renaming algorithm is still much larger than the ideal range $[1, n]$. We shall show in the next section that, instead, whenever mediated through the RC-simulation, this renaming algorithm uses a range of names that shrinks as the partial asynchrony decreases, up to $[1, n]$ for partial asynchrony zero.

Observation 1 *Even if the system is 0-partially asynchronous, there is an execution of the renaming algorithm in* [9] *for which the range of names is* $[1, \frac{3n}{2}]$.

RC-simulation mediated long-lived renaming. In the following, we show that, through RC-simulation, the renaming algorithm of [9] adapts gracefully to partial asynchrony, while Observation 1 shows that the performances of this algorithm alone do not adapt to partial asynchrony. The result below is a corollary of Theorem 1. In particular, it shows that when the system is synchronous, or runs in lock-steps, the renaming algorithm mediated by the RC-simulation provides perfect renaming, using names in the range $[1, n]$. More generally, the mediation through RC-simulation of the renaming algorithm in [9] produces names in a range that grows linearly with partial asynchrony k.

Theorem 2. *If the system is k-partially asynchronous, for $0 \leq k < n$, then the range of names provided by the renaming algorithm in* [9] *mediated by the RC-simulation is* $[1, n + k]$.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sept. 1993.
2. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Partial synchrony based on set timeliness. *Distributed Computing*, 25(3):249–260, 2012.
3. M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, Apr. 2011.
4. M. K. Aguilera and S. Toueg. Adaptive progress: a gracefully-degrading liveness property. *Distributed Computing*, 22(5-6):303–334, 2010.
5. D. Alistarh, S. Gilbert, R. Guerraoui, and C. Travers. Generating fast indulgent algorithms. *Theory Comput. Syst.*, 51(4):404–424, 2012.
6. H. Attiya. Adapting to point contention with long-lived safe agreement. In *SIROCCO'06*, pages 10–23. Springer-Verlag.
7. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
8. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524548, 1990.
9. H. Attiya and J. Welch. *Distributed Computing Fundamentals, Simulations, and Advanced Topics, Second Edition*. John Wiley and Sons, Inc., 2004.
10. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC '93*, pages 91–100. ACM, 1993.
11. E. Borowsky and E. Gafni. The Implication of the Borowsky-Gafni Simulation on the Set-Consensus Hierarchy. Technical report, UCLA, 1993.
12. E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

13. J. E. Burns and G. L. Peterson. The ambiguity of choosing. PODC '89, pages 145–157. ACM, 1989.
14. A. Castañeda, D. Imbs, S. Rajsbaum, and M. Raynal. Renaming is weaker than set agreement but for perfect renaming: A map of sub-consensus tasks. LATIN'12, pages 145–156. Springer-Verlag, 2012.
15. A. Castañeda, S. Rajsbaum, and M. Raynal. The renaming problem in shared memory systems: An introduction. *Comput. Sci. Rev.*, 5(3):229–251, Aug. 2011.
16. T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies (extended abstract). In *PODC '94*, pages 334–343. ACM, 1994.
17. S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.
18. S. Chaudhuri and P. Reiners. Understanding the Set Consensus Partial Order Using the Borowsky-Gafni Simulation (Extended Abstract). In *WDAG'96*, pages 362–379. Springer-Verlag, 1996.
19. F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, June 1999.
20. D. Dolev. The byzantine generals strike again. *J. of Algorithms*, 3(1):14 – 30, 1982.
21. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.
22. M. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility Of Distributed Commit With One Faulty Process. *Journal of the ACM*, 32(2), Apr. 1985.
23. M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *FCT'83*, pages 127–140. Springer-Verlag, 1983.
24. E. Gafni. The extended BG-simulation and the characterization of t-resiliency. In *STOC '09*, pages 85–92. ACM, 2009.
25. E. Gafni and R. Guerraoui. Generalized universality. In *CONCUR 2011*, volume 6901 of *LNCS*, pages 17–27. Springer Berlin Heidelberg, 2011.
26. R. Guerraoui. Indulgent algorithms. In *PODC*, pages 289–297. ACM, 2000.
27. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.
28. M. Herlihy and E. Ruppert. On the existence of booster types. In *FOCS '00*, pages 653 – 663. IEEE Computer Society, 2000.
29. D. Imbs, S. Rajsbaum, and M. Raynal. The universe of symmetry breaking tasks. In *SIROCCO'11*, pages 66–77. Springer-Verlag, 2011.
30. D. Imbs and M. Raynal. Visiting Gafni's Reduction Land: From the BG Simulation to the Extended BG Simulation. In *SSS '09*, pages 369–383. Springer-Verlag, 2009.
31. D. Imbs and M. Raynal. The multiplicative power of consensus numbers. In *PODC '10*, pages 26–35. ACM, 2010.
32. I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: Preliminary version. *SIGACT News*, 32(2):45–63, June 2001.
33. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
34. L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
35. W.-K. Lo and V. Hadzilacos. On the power of shared object types to implement one-resilient Consensus. *Distributed Computing*, 13(4):219–238, Nov. 2000.
36. N. Lynch and S. Rajsbaum. On the Borowsky-Gafni Simulation Algorithm. In *ISTCS '96*, pages 4–15. IEEE Computer Society, June 1996.
37. F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.